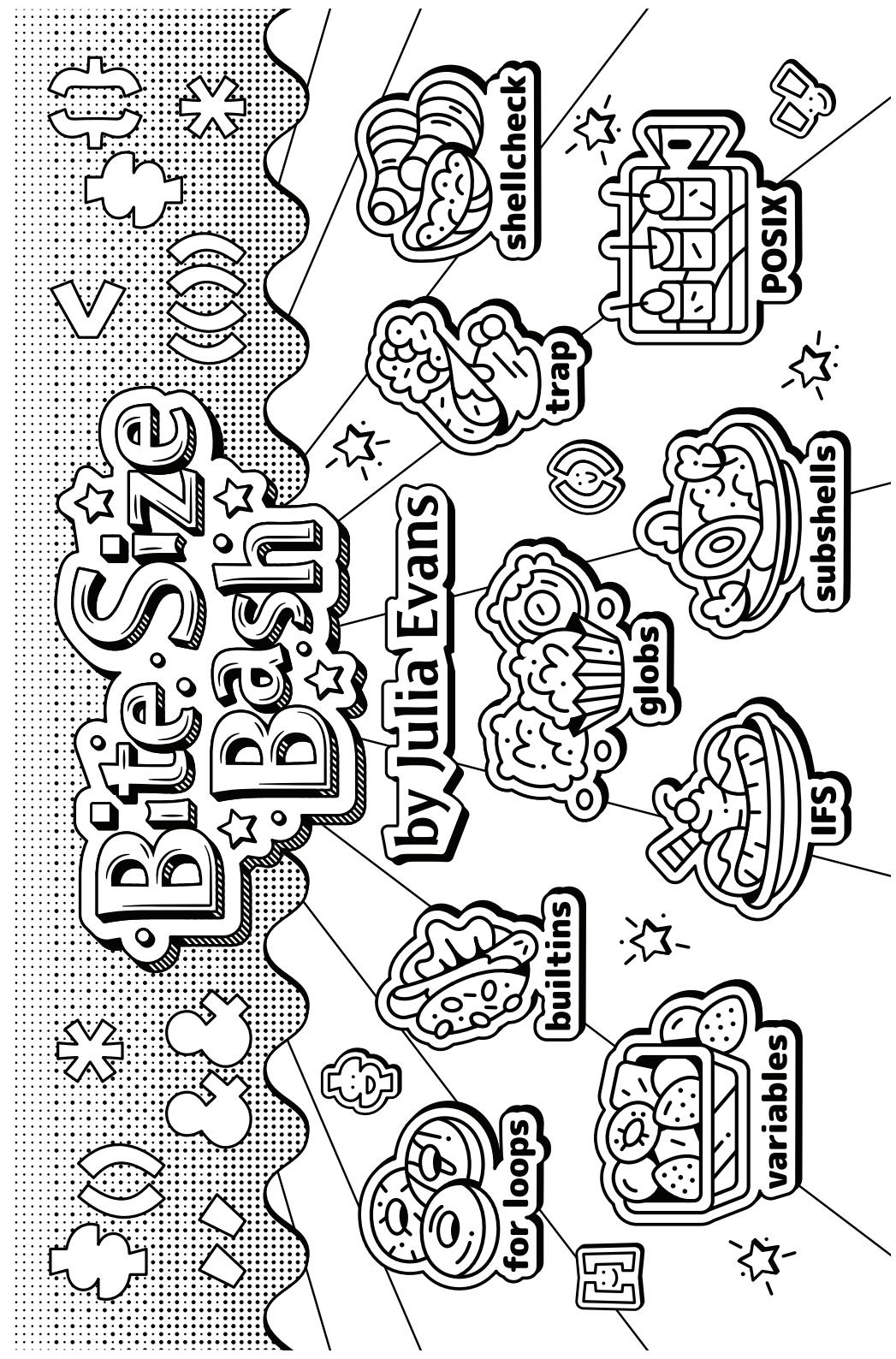
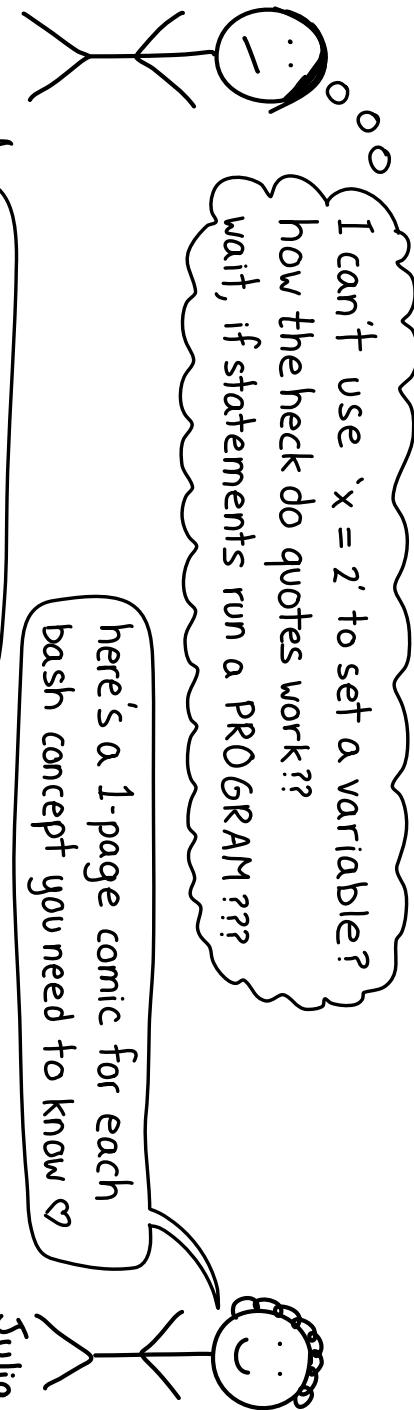


* wizardzines.com *



hello! we're here because bash* is a very weird programming language.



*most of this zine also applies to other shells, like zsh

thanks for reading

There's more to learn about bash than what's in this zine, but I've written a lot of bash scripts and this is all I've needed so far. If the task is too complicated for my bash skills, I just use a different language.

two pieces of parting advice:

- ① when your bash script does something you don't understand, figure out why!
→ ok, this is my advice for literally all programming "

- ② use shellcheck! And read the shellcheck wiki when it tells you about an error :)

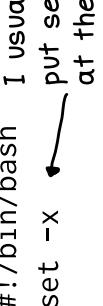
credits

Cover art: Vladimir kašković
Editing: Dolly Lanuza, Kamal Marhubi
Copy Editing: Courtney Johnson
and thanks to all 11 beta readers ♥

debugging

26

our hero: set -x

set -x prints out every line of a script as it executes, with all the variables expanded!
#!/bin/bash
set -x 

or bash -x

\$ bash -x script.sh
does the same thing as putting set -x at the top of script.sh

you can stop before every line

```
trap read DEBUG  
↑  
the DEBUG "signal"  
is triggered before  
every line of code
```

a fancy step debugger trick

put this at the start of your script to confirm every line before it runs:

```
trap '(read -p "[${BASH_SOURCE:$LINENO}] $BASH_COMMAND")' DEBUG  
↑  
read -p prints a script line  
message, press filename number enter to continue  
next command that will run
```

you can stop before every line

```
trap read DEBUG  
↑  
the DEBUG "signal"  
is triggered before  
every line of code
```

how to print better error messages

```
this die function:  
die() { echo $1 >&2; exit 1; }  
lets you exit the program  
and print a message if a  
command fails, like this:  
some_command || die "oh no!"
```

table of contents

basics

why I ♡ bash	4
POSIX	5
shellcheck	6
variables	7
env variables	8
arguments	9
builtins	10
quotes	11
globs	12
redirects	13

cheat sheets (in the middle!)

brackets	14
non-POSIX features	15
\$() \${{}}	16
bash >	17
\$1 <	18
pipes	19
parameter expansion	20
background processes	22
subshells	23
trap	24
errors	25
debugging	26

getting fancy

if statements	16
for loops	17
reading input	18
functions	19
pipes	20
parameter expansion	21
background processes	22
subshells	23
trap	24
errors	25
debugging	26

Why I ❤️ bash

4

it's SO easy to get started

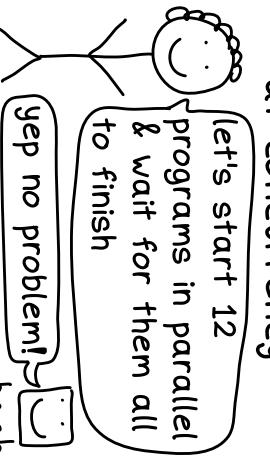
Here's how:

① Make a file called hello.sh and put some commands in it, like

ls /tmp

② Run it with bash hello.sh

it's surprisingly good at concurrency

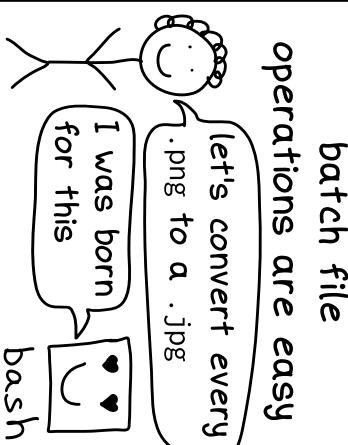


pipes & redirects are super easy
managing pipes in other languages is annoying. in bash, it's just:

cmd1 | cmd2

♥ it doesn't change ♥

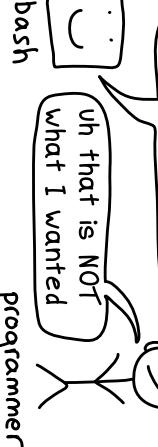
bash is weird and old, but the basics of how it works haven't changed in 30 years. If you learn it now, it'll be the same in 10 years.

batch file operations are easy


errors

by default, bash will continue after errors

that program's exit status was 1? who cares, let's keep running!!!



by default, unset variables don't error

rm -r "\$HOME/\$SOMEPTH"





set -e stops the script on errors

set -e

this makes your scripts WAY more predictable

set -o pipefail makes the pipe fail if any command fails

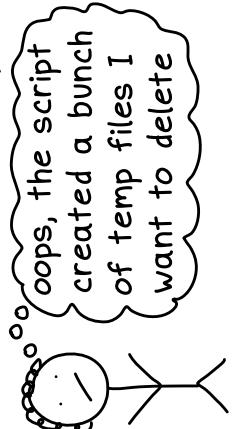
you can combine set -e, set -u, and set -o pipefail into one command I put at the top of all my scripts:
` ; set -euo pipefail`

25

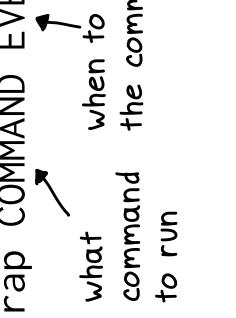
trap

24

when your script exits, sometimes you need to clean up
 oops, the script created a bunch of temp files I want to delete



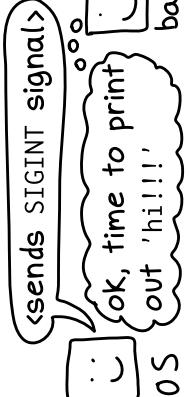
trap sets up callbacks
 trap COMMAND EVENT
 what command to run to run
 when to run the command



events you can trap
 → unix signals (INT, TERM, etc)
 → the script exiting (EXIT)
 → every line of code (DEBUG)
 → function returns (RETURN)

example: Kill all background processes when Ctrl+C is pressed
 trap 'kill \$(jobs -p)' INT
 important: single quotes!
 when you press CTRL+C, the OS sends the script a SIGINT signal

bash runs COMMAND when EVENT happens
 trap "echo 'hi!!!'" INT
 (sends SIGINT signal)
 ok, time to print out 'hi!!!'
 OS bash



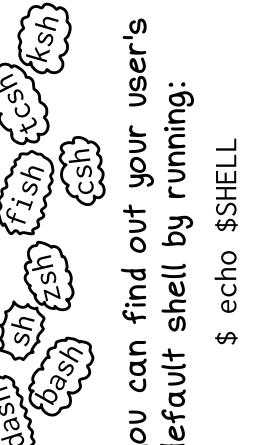
example: cleanup files when the script exits
 function cleanup() {
 rm -rf \$TEMPDIR
 rm \$TEMPFILE
 }
 trap cleanup EXIT

5

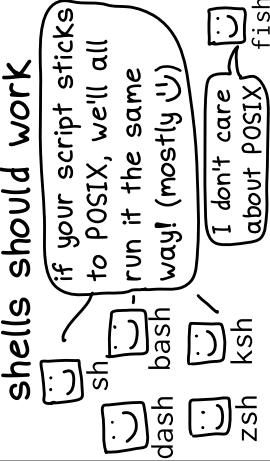
POSIX compatibility

there are lots of Unix shells
 dash sh zsh bash tcsh csh ksh fish

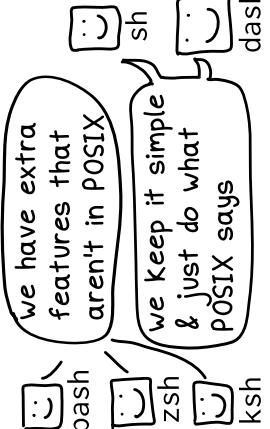
you can find out your user's default shell by running:
 \$ echo \$SHELL



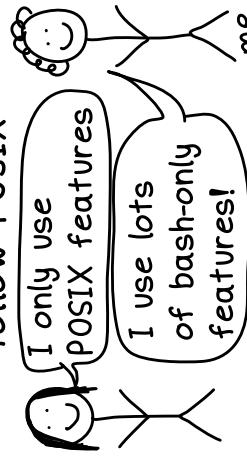
POSIX is a standard that defines how Unix shells should work
 if your script sticks to POSIX, we'll all run it the same way! (mostly :))
 dash bash zsh ksh fish
 I don't care about POSIX



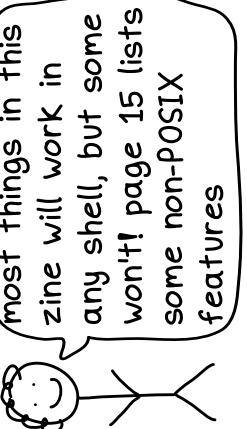
some shells have extra features
 we have extra features that aren't in POSIX
 sh zsh ksh dash
 we keep it simple & just do what POSIX says



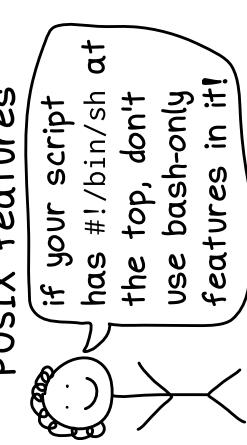
this zine is about bash scripting
 most things in this zine will work in any shell, but some won't! page 15 lists some non-POSIX features



some people write all their scripts to follow POSIX
 I only use POSIX features
 I use lots of bash-only features!



on most systems, /bin/sh only supports POSIX features
 if your script has #!/bin/sh at the top, don't use bash-only features in it!



Shellcheck

6

shellcheck finds problems with your shell scripts

```
$ shellcheck my-script.sh
```

oops, you can't use ~ in an if [...]!

shellcheck

it even tells you about misused commands

hey, it looks like you're not using grep correctly here

wow, I'm not! thanks!

shellcheck

it checks for hundreds of common shell scripting errors

hey, that's a bash only feature but your script starts with #!/bin/sh

shellcheck

your text editor probably has a shellcheck plugin

I can check your shell scripts every time you save!

shellcheck

Subshells

23

a subshell is a child shell process

hey, can you run this bash code for me?

sure thing! other bash process

shellcheck

some ways to create a subshell

① put code in parentheses (...)

(cd \$DIR; ls)

runs in subshell

② put code in \$(...)

var=\$(cat file.txt)

runs in subshell

③ pipe/redirect to a code block cat x.txt | while read line...

piping to a loop makes the loop run in a subshell

④ + lots more for example, process substitution <() creates a subshell

cd in a subshell doesn't cd in the parent shell

(cd subdir/

mv x.txt y.txt)

I like to do this so I don't have to remember to cd back at the end!

setting a variable in a subshell doesn't update it in the main shell

var=3

(var=2)

echo \$var

this prints

it's easy to create a subshell and not notice x=\$(some_function) I changed directories in some_function, why didn't it work? It's running in a subshell!

every shellcheck error has a number (like "SC2013")

and the shellcheck wiki has a page for every error, with examples! I've learned a lot from the wiki.

background processes

22

scripts can run many processes in parallel

```
python -m http.server &  
curl localhost:8080
```

& starts python in the "background", so it keeps running while curl runs

wait waits for all background processes to finish

```
command1 &  
command2 &
```

wait

this waits for both command1 and command2 to finish

background processes sometimes exit when you close your terminal you can keep them running with nohup or by using tmux/screen.

```
$ nohup ./command &
```

concurrency is easy* in bash
in other languages:

```
thing1 &  
thing2 &  
wait
```

*(if you keep it very simple)

jobs list shell's background processes fg and bg move process to foreground/background

disown like nohup, but after process has started

move process to foreground/background

jobs, fg, bg, and disown let you juggle many processes in the same terminal, but I almost always just use multiple terminals instead

7

variables

how to use a variable:

```
"$var"  
filename=blah.txt  
echo "$filename"
```

they're case sensitive.
environment variables are traditionally all-caps, like \$HOME

how to set a variable

right! (no spaces!)
var=value
var = value → wrong
var = value will try to run the program var with the arguments "=" and "value"

always use quotes around variables

```
$ filename="swan 1.txt"  
$ cat "$filename"  
ok, I'll run  
cat "swan 1.txt"  
oh no! we didn't mean that!  
um swan and 1.txt  
cat  
bash  
don't exist...
```

```
 ${varname}
```

To add a suffix to a variable like "2", you have to use \${varname}. Here's why:
prints "",
zoo2 isn't a variable
\$ echo "\$zoo2"
\$ echo "\${zoo}2" this prints "panda2" like we wanted

Environment variables

8

every process has environment variables
printing out your shell's environment variables is easy, just run:

```
$ env
```

child processes inherit environment variables
this is why the variables set in your .bashrc are set in all programs you start from the terminal.
They're all child processes of your bash shell!

shell scripts have 2 kinds of variables

1. environment variables
2. shell variables

unlike in other languages, in bash you access both of these in the exact same way: \$VARIABLE

shell variables aren't inherited

var=panda

\$var only gets set in this process, not in child processes

you can set env vars when starting a program
2 ways to do it (both good!):

- ① \$ env VAR=panda ./myprogram
- ② \$ VAR=panda ./myprogram

ok! I'll set VAR to panda and then start ./myprogram

(here bash sets VAR=panda)

\${}: "parameter expansion" 2)

{} is really powerful

it can do a lot of string operations!

my favorite is search/replace.

see page 7 for when to use this instead of \$var

\$#var

length of the string or array var

remove the prefix/suffix pattern from var. Example:

\$var#pattern

\$var%pattern

/ replaces first instance, // replaces every instance, search & replace example:

\$ x="I'm a bearbear!"

\$ echo \${x%.*}

I'm a pandabear!

use a default value like \$othervar if var is unset/null

{var:?some error}
prints "some error" and exits if var is unset/null

get a substring of var

\$var:offset:length

there are LOTS more, look up "bash parameter expansion"!

\$var#pattern

\$var%pattern

remove the prefix/suffix pattern from var. Example:

\$ x=motorcycle.svg

\$ echo "\${x%.svg}"

motorcycle

Pipes

20

sometimes you want to send the output of one process to the input of another

```
$ ls | wc -l  
53 ↪ 53 files!
```

a pipe is a pair of 2 magical file descriptors

```
|s|-----> IN -----> pipe -----> OUT -----> WC  
          |           |           |           |  
          file descriptors
```

when ls does

```
write(IN, "hi")  
wc can read it!  
read(OUT) → "hi"  
Pipes are one way. →  
You can't write to OUT
```

the OS creates a buffer for each pipe

IN (data waiting OUT to be read)

when the buffer gets full:

write(IN, "...")

it's full! I'm going to pause you until process there's room again

named pipes

you can create a file that acts like a pipe with mkfifo

```
$ mkfifo mypipe  
$ ls > mypipe &  
$ wc < mypipe
```

this does the same thing as ls | wc

you can use pipes in other languages!

only shell has the syntax

```
process1 | process2
```

but you can create pipes in basically any language!

arguments

Q

get a script's arguments with \$0, \$1, \$2, etc

```
$ svg2png old.svg new.png  
$0 is "svg2png" "old.svg" "new.png"  
$1 is (script's name)  
$2 is
```

arguments are great for making simple scripts

Here's a 1-line svg2png script I use to convert SVGs to PNGs:

```
#!/bin/bash  
inkscape "$1" -b white --export-png="$2"  
I run it like this:  
$ svg2png old.svg new.png
```

"\$@": all arguments

\$@ is an array of all the arguments except \$0.

This script passes all its arguments to ls --color:

```
#!/bin/bash  
ls --color "$@"
```

you can loop over arguments

```
for i in "$@"  
do  
    ...  
done
```

in our svg2png example, this would loop over old.svg and new.png

shift removes the first argument

this prints the script's first argument

```
echo $1  
shift  
echo $1
```

this prints the second argument

builtins

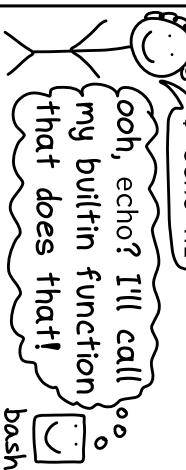
most bash commands are programs

You can run which to find out which binary is being used for a program:

```
$ which ls  
/bin/ls
```

examples of builtins

`type` `Source`
`alias` `declare`
`read` `cd`
`printf` `echo`



but some commands are functions inside the bash program

alias lets you set up shorthand commands, like:

```
alias gc="git commit"
```

~/.bashrc runs when bash starts, put aliases there!

a useful builtin:

`alias`

`shorthand` commands, like:

```
alias gc="git commit"
```

~/.bashrc runs when bash starts, put aliases there!

type tells you if a command is a builtin

```
$ type grep  
grep is /bin/grep  
$ type echo  
echo is a builtin  
$ type cd  
cd is a builtin
```

a useful builtin:

`source`

bash script.sh runs script.sh in a subprocess, so you can't use its variables / functions. source script.sh is like pasting the contents of script.sh

functions

you can't return a string

defining functions is easy

```
say_hello() {  
    echo "Hello!"  
}
```

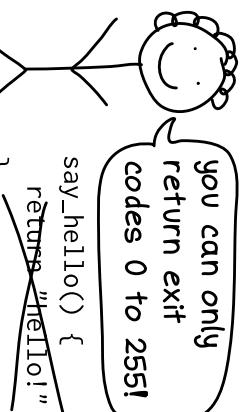
... and so is calling them

```
say_hello← no parentheses!
```

functions have exit codes

```
failing_function()  
{  
    return 1  
}
```

0 is success, everything else is a failure. A program's exit codes work the same way.



19

the local keyword declares local variables

```
say_hello() {  
    echo "Hello $1!"  
}  
say_hello "Ahmed"  
↑  
not say_hello("Ahmed")!
```

arguments are \$1, \$2, \$3, etc

```
say_hello() {  
    echo "Hello $1!"  
}  
say_hello "Ahmed"
```

local x=y=\$((date))

```
say_hello() {  
    echo "Hello $1!"  
}  
say_hello "Ahmed"
```

local x=\$asdf

```
say_hello() {  
    echo "Hello $1!"  
}  
say_hello "Ahmed"
```

never fails, even if asdf doesn't exist

```
say_hello() {  
    echo "Hello $1!"  
}  
say_hello "Ahmed"
```

this one will fail

```
say_hello() {  
    echo "Hello $1!"  
}  
say_hello "Ahmed"
```

I have NO IDEA why it's like this, bash is weird sometimes

reading input

18

read -r var
reads stdin into a variable

```
$ read -r greeting
    ahmed fatima
$ echo "$greeting"
    fatima
hello there!
```

you can also read into multiple variables

```
$ read -r name1 name2
    ahmed fatima
$ echo "$name2"
    fatima
```

by default, read strips whitespace

```
" a b c " -> "a b c"
```

it uses the IFS ("Input Field Separator") variable to decide what to strip

set IFS='' to avoid stripping whitespace

empty string

```
$ IFS=''; read -r greeting
    hi there!
$ echo "$greeting"
    hi there!
    ↪ the spaces are still there!
```

more IFS uses: loop over every line of a file

by default, for loops will loop over every word of a file (not every line). Set IFS=' ' to loop over every line instead!

```
IFS=' '
    ↪ for line in $(cat file.txt)
do
    echo $line
done!
```

single quotes always give you exactly what you typed in

quotes

double quotes expand variables, single quotes don't

```
$ echo 'home: $HOME'
    $ echo "home: $HOME"
    ↪
    ↪ $HOME got expanded to /home/bork
```

single quotes always give you exactly what you typed in

you can quote multiline strings

```
$ MESSAGE="Usage:
```

here's an explanation of how to use this script!"

how to concatenate strings

put them next to each other!

```
$ echo "hi ""there"
    ↪
    ↪ x + y doesn't add strings:
    $ echo "hi" ± " there"
    ↪ hi ± there
```

escaping ' and "

here are a few ways to get a ' or ":

- ' and \'
- "" and "",
- \$'\'',
- "\\"",

a trick to escape any string: !:q:p

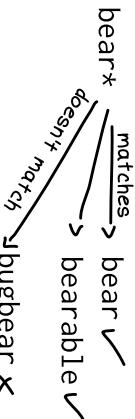
get bash to do it for you!

```
$ # He said "that'\''s $5"
    $ !:q:p
    ↪
    ↪ '# He said "that'\''s $5"
        this only works in bash, not zsh.
        ! is an "event designator" and
        :q:p is a "modifier"
```

Glob

globs are a way to match strings

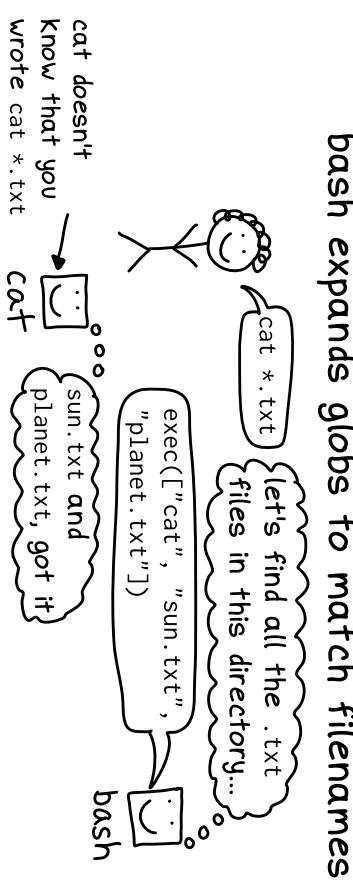
beware: the * and the ? in a glob are different than * and ? in a regular expression!!!



there are just 3 special characters

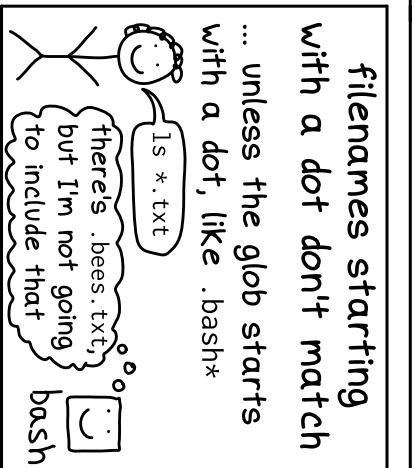
- * matches 0+ characters
- ? matches 1 character
- [abc] matches a or b or c

I usually just use * in my globs



use quotes to pass a literal '*' to a command

\$ egrep 'b.*' file.txt
the regexp 'b.*' needs to be quoted so that bash won't translate it into a list of files with b. at the start



for loops

for loop syntax

the semicolons are weird
usually in bash you can always replace a newline with a semicolon. But not with for loops!

for i in a b; do ...; done
you need semicolons before do and done but it's a syntax error to put one after do

for loops loop over words, not lines
for word in \$(cat file.txt)
loops over every word in the file, NOT every line (see page 18 for how to change this!)

how to loop over a range of numbers

3 ways:
for i in \$(seq 1 5)
for i in {1..5}
for ((i=1; i<6; i++))
these two only work in bash, not sh

while loop syntax
while COMMAND
do
...
done
like an if statement, runs COMMAND and checks if it returns 0 (success)

if statements

16

in bash, every command has an exit status
0 = success
any other number = failure
bash puts the exit status of the last command in a special variable called \$?:

why is 0 success?
there's only one way to succeed, but there are LOTS of ways to fail. For example
grep THING x.txt
will exit with status:
1 if THING isn't in x.txt
2 if x.txt doesn't exist

there are 2 commands often used in if statements: [and [[
if [-e file.txt] /usr/bin/[(aka test) is a program* that returns 0 if the test you pass it succeeds

*in bash, [is a builtin that acts like /usr/bin/[

bash if statements test if a command succeeds
if COMMAND; then
do a thing
fi:
① runs COMMAND
② if COMMAND returns 0 (success), then do the thing

true

true is a command that always succeeds, not a boolean

combine with && and ||

if [-e file1] && [-e file2]

man test for more on [

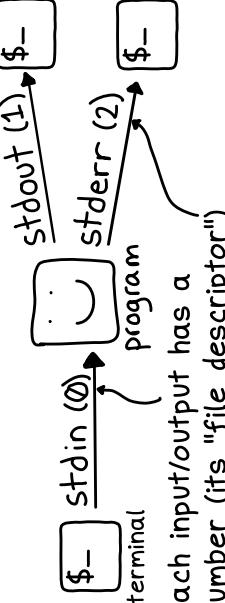
you can do a lot!

13

> redirects <

unix programs have 1 input and 2 outputs

When you run a command from a terminal, they all go to/from the terminal by default.
< redirects stdin
\$ wc < file.txt
\$ cat file.txt | wc
these both read file.txt to wc's stdin each input/output has a number (its "file descriptor")



> redirects stdout
\$ cmd > file.txt
> redirects stderr
\$ cmd 2> file.txt

2>&1 redirects stderr to stdout

\$ cmd > file.txt 2>&1
\$ cmd > /dev/null
\$ cmd > file.txt
\$ cmd 2> /dev/null
\$ cmd 2>&1

/dev/null
your operating system ignores all writes to /dev/null.

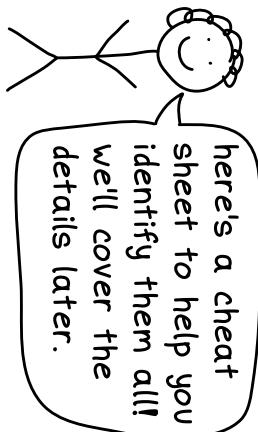
sudo doesn't affect redirects
your bash shell opens a file to redirect to it, and it's running as you. So
\$ sudo echo x > /etc/xyz
won't work. do this instead:
\$ echo x | sudo tee /etc/xyz

< redirects stdin
\$- < file.txt
> redirects stdout
\$- > file.txt
> redirects stderr
\$- 2> file.txt

brackets cheat sheet

14

shell scripts have a lot of brackets



here's a cheat sheet to help you identify them all! we'll cover the details later.

(cd ~/music; pwd)

(...) runs commands in a subshell.

{ cd ~/music; pwd }

{...} groups commands. runs in the same process.

x=(1 2 3)

x=(...) creates an array

x=\$((2+2))

\$() does arithmetic

/usr/bin/[is a program that evaluates statements

if [[...]]

[[] is bash syntax. it's more powerful than [

<(COMMAND)

"process substitution": an alternative to pipes

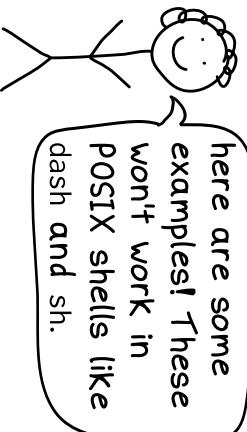
\$(var//search/replace)

see page 21 for more about \${...}!

non-POSIX features

15

some bash features aren't in the POSIX spec



POSIX alternative:

[...]

diff <./cmd1> <./cmd2>

this is called "process substitution", you can use named pipes instead

the local keyword

in POSIX shells, all variables are global

\$'\\n'

POSIX alternative:

\$(printf "\\n")

[[\$DIR = /home/*]]

for ((i=0; i <3; i++))

sh only has for x in ... loops, not C-style loops

POSIX alternative: pipe

match strings with grep

VAR=\$(cat file.txt)

\$COMMAND is equal to COMMAND's stdout